# Ficha 4

Programação Imperativa

## 1 Árvores

1. Considere a seguinte definição de um tipo para representar árvores binárias de inteiros.

```
typedef struct nodo *ABin;

struct nodo {
    int valor;
    ABin esq, dir;
};
```

   (a) Considere a seguinte definição de uma função que cria uma lista ligada de inteiros a partir de uma travessia *inorder* de uma árvore binária:

```
LInt inorder (ABin a) {
  Lint r, aux;
  if (a == NULL) r = NULL;
  else { r = (LInt) malloc (sizeof (struct slist));
         r->valor = a->valor;
         r->prox = inorder (a->dir);
         aux = inorder (a->esq);
         r = concat (aux,r);
       }
  return r;
}
```

   De forma a optimizar esta função vamos apresentar uma definição alternativa que não usa a função `snoc` de listas. Esta alternativa passa por usar uma função auxiliar que insere à cabeça de uma lista (inicialmente vazia) os vários elementos da árvore.

```
LInt inorder (ABin a) {
    return (inorderAcc (a, NULL));
}

LInt inorderAcc (ABin a, LInt l) {
  LInt r;
  if (a == NULL) r = l;
  else { r = (LInt) malloc (sizeof(struct slist));
```

```
            r->valor = a->valor;
            r->prox = inorderAcc (a->dir, l);
            r = inorderAcc (a->esq, r);
          }
      return r;
}
```

Apresente definições das funções `LInt preorder (ABin a)` e `LInt posorder (ABin a)` que criam listas a partir das travessias *preorder* e *posorder* de uma árvore binária sem usar a função `concat`

(b) Defina uma função `ABin remove (ABin a, int x)` que remove um elemento de uma árvore binária de procura.

2. Uma das aplicações mais frequentes de árvores binárias de procura é na implementação de funções finitas. Uma função finita é um conjunto de pares (chave,informação) em que cada chave não aparece repetida. Nos casos em que existe uma ordem sobre as chaves pode-se então usar uma árvore binária de procura (ordenada pela chave) para guardar uma destas funções finitas.

   As operações que devem estar disponíveis são:

   - adicionar um novo par (chave,informação)
   - remover o par associado a uma dada chave
   - modificar a informação associada a um dado par
   - procura da informação associada a uma dada chave

   Considere a seguinte definição para implementar funções finitas em que as chaves são *strings* e a informação associada a cada chave é um endereço de memória.

```
typedef struct fmap {
    char *key;
    void *info;
    struct fmap *left, *right;
} Node, *Fmap;
```

   e implemente as seguintes funções:

   (a) `Fmap addPair (Fmap f, char *k, void *i)` que adiciona um novo par `(k,i)` à função finita `f`. A função deverá retornar `NULL` caso a operação não seja possível (por exemplo porque a chave `k` já existe em `f`).

   (b) `int remove (Fmap *f, char *k, int *r)` que remove a chave `k` da função `*f`. A função retorna um código de erro (`0` no caso de sucesso). Note que é passado como argumento à função o endereço da árvore – trata-se de uma parâmetro de entrada/saída.

   (c) `Fmap udpate (FMap f, char *k, void *i)` que modifica a informação associada a `k` para `i`. Se a chave `k` ainda não existir em `f`, deverá ser acrescentado o par `(k,i)`.

(d) `void *lookup (Fmap f, char *k)` que calcula a informação associada a `k` na função `f`. A função deverá retornar `NULL` caso a chave `k` não exista.

3. Suponha que para resolver o problema descrito na Ficha anterior se optou por usar uma árvore binária de procura (ordenada pelo número do aluno) vez de um array.

   (a) Defina os novos tipos de dados para esta implementação.

   (b) Apresente definições das funções `acrescentaAluno`, `procura` e `aprovados` para esta nova implementação.
   Tenha o cuidado de rever os tipos destas funções nesta nova implementação.

   (c) Defina uma função que liberte o espaço ocupado por uma destas árvores.

4. Suponha agora que se pretende implementar uma nova funcionalidade: listagem das notas finais dos alunos, por ordem crescente do seu nome. Faça as alterações necessárias para que se tenha em qualquer altura acesso ordenado (por ordem crescente do nome) à lista dos alunos.

   Implemente a referida função de listagem (para o ecran).

5. Considere a seguinte definição:

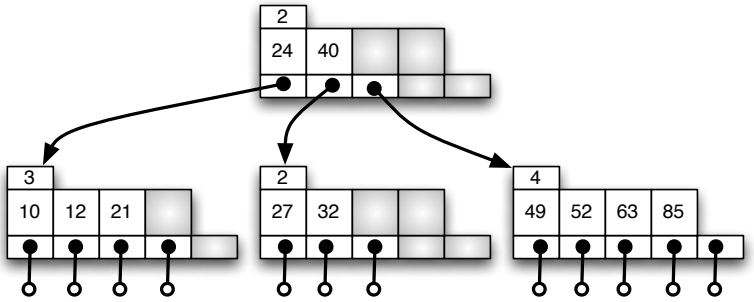
```
typedef struct node {
   int t;
   int valores [N-1];
   struct node *menores [N]
   struct {
       int valor;
       struct node * menores;
       } tab [N];
} *ArvB;
```

   Esta definição pode ser usada para representar **árvores B** de ordem `N`.

   Em cada nodo de uma destas estruturas guardam-se até `N-1` items (neste caso, inteiros). Daí que cada nodo contenha um inteiro `t` que indica o número de items que estão a ser guardados nesse nodo.

   Para além desse inteiro, cada nodo contem um array onde são guardados os vários items e outro onde se guardam os endereços das árvores onde se encontram os items menores ou iguais a esse item. O último elemento deste último array contem o endereço da árvore onde se encontram os items maiores do que todos os items deste nodo.

   Veja-se por exemplo a seguinte árvore de altura 2 e ordem 5.



Apresente definições das seguintes funções:

(a) `int quantos (ArvB a)` que calcula quantos elementos tem uma árvore.

(b) `int existe (ArvB a, int x)` que testa se um dado inteiro pertence a uma árvore (retorna `0` sse não existir).

(c) `void imprime (ArvB a)` que imprime no ecran os elementos de uma destas árvores por ordem crescente.

(d) `int maior (ArvB a)` que calcula o maior elemento de uma árvore não vazia.